Data Stream Queries to Apache SPARK

Michelle Brundin
Abstract

Data Stream Queries to Apache SPARK

Michelle Brundin

Many fields have a need to process and analyze data streams in real-time. In industrial applications the data can come from big sensor networks, where the processing of the data streams can be used for performance monitoring and fault detection in real time. Another example is in social media where data stream processing can be used to detect and prevent spam. A data stream management system (DSMS) is a system that can be used to manage and query continuously received data streams. The queries a DSMS executes are called continuous queries (CQs). In contrast to regular database queries they execute continuously until canceled. SCSQ is a DSMS developed at Uppsala university. Apache Spark is a large scale general data processing engine. It has, among other things, a component for data stream processing, Spark Streaming. In this project a system called SCSQ Spark Streaming Interface (SSI) was implemented that allows Spark Streaming applications to be called from CQs in SCSQ. It allows the Spark Streaming applications to receive input streams from SCSQ as well as emitting resulting stream elements back to SCSQ. To demonstrate SSI, two examples are shown where it is used for stream clustering in CQs using the streaming k-means implementation in Spark Streaming.
## Content

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>Background</td>
<td>9</td>
</tr>
<tr>
<td>2.1</td>
<td>Apache Spark</td>
<td>9</td>
</tr>
<tr>
<td>2.1.1</td>
<td>Resilient Distributed Datasets, RDDs</td>
<td>9</td>
</tr>
<tr>
<td>2.1.2</td>
<td>Spark Streaming</td>
<td>10</td>
</tr>
<tr>
<td>2.1.3</td>
<td>MLlib</td>
<td>11</td>
</tr>
<tr>
<td>2.1.4</td>
<td>Programming model</td>
<td>12</td>
</tr>
<tr>
<td>2.1.5</td>
<td>Example Spark Streaming Application</td>
<td>13</td>
</tr>
<tr>
<td>2.2</td>
<td>SCSQ</td>
<td>15</td>
</tr>
<tr>
<td>2.2.1</td>
<td>Amos</td>
<td>15</td>
</tr>
<tr>
<td>2.2.2</td>
<td>Connecting to external systems</td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td>SCSQ Spark Streaming Interface</td>
<td>19</td>
</tr>
<tr>
<td>3.1</td>
<td>Overview</td>
<td>19</td>
</tr>
<tr>
<td>3.2</td>
<td>K-Means in MLlib</td>
<td>20</td>
</tr>
<tr>
<td>3.3</td>
<td>Off-line stream clustering</td>
<td>21</td>
</tr>
<tr>
<td>3.4</td>
<td>On-line stream clustering</td>
<td>23</td>
</tr>
<tr>
<td>4</td>
<td>Implementation</td>
<td>26</td>
</tr>
<tr>
<td>4.1</td>
<td>The sparkStream foreign function</td>
<td>26</td>
</tr>
<tr>
<td>4.2</td>
<td>The SCSQSparkContext class</td>
<td>28</td>
</tr>
<tr>
<td>4.3</td>
<td>Supported data types</td>
<td>29</td>
</tr>
<tr>
<td>4.4</td>
<td>SCSQ Receiver</td>
<td>30</td>
</tr>
<tr>
<td>4.5</td>
<td>SCSQ Sink</td>
<td>31</td>
</tr>
<tr>
<td>4.6</td>
<td>The SparkSubmitter</td>
<td>32</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>4.7</td>
<td>Stream clustering</td>
<td>33</td>
</tr>
<tr>
<td>4.7.1</td>
<td>Off-line stream clustering</td>
<td>33</td>
</tr>
<tr>
<td>4.7.2</td>
<td>On-line stream clustering</td>
<td>35</td>
</tr>
<tr>
<td>5</td>
<td>Evaluation</td>
<td>37</td>
</tr>
<tr>
<td>5.1</td>
<td>Stream clustering measurements</td>
<td>37</td>
</tr>
<tr>
<td>5.1.1</td>
<td>On-line stream clustering</td>
<td>37</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Off-line stream clustering</td>
<td>38</td>
</tr>
<tr>
<td>5.2</td>
<td>Interface performance</td>
<td>40</td>
</tr>
<tr>
<td>5.2.1</td>
<td>SCSQ Receiver performance</td>
<td>40</td>
</tr>
<tr>
<td>5.2.2</td>
<td>SCSQ Sink performance</td>
<td>41</td>
</tr>
<tr>
<td>6</td>
<td>Conclusions</td>
<td>43</td>
</tr>
<tr>
<td>7</td>
<td>Future Work</td>
<td>44</td>
</tr>
<tr>
<td>Appendix A</td>
<td>Java and SCSQL code for off-line stream clustering</td>
<td>45</td>
</tr>
<tr>
<td>Appendix B</td>
<td>Java and SCSQL code for on-line stream clustering</td>
<td>47</td>
</tr>
</tbody>
</table>
1 Introduction

The need to process and analyze big data streams is a problem found in many different fields. In industrial applications it can be very important to analyze data streams from big sensor networks in real-time. Performance monitoring, fault detection and fault prediction is some of the reasons this is valuable in an industrial context. A machine equipped with sensors will, when in operation, generate big amounts of data. By having a system that continuously analyzes this data in real-time it can in an early stage be discovered when the equipment is not working as intended. Measures can then be taken before the problem gets any bigger. Other areas where real-time data stream processing can be important is for example online social networks to detect spam in real-time. Also in scientific research data stream processing can be an important tool. Experiments might generate so much data that it is not possible to store all the data. Then the data stream must be processed immediately in real-time.

In these types of applications, a traditional database management system (DBMS) cannot be used since it only works with stale data that has to be loaded and indexed before it can be analyzed. To process the data stream a data stream management system (DSMS) can be used. As the name suggests, a DSMS is a system that can manage and query continuously received data streams. At Uppsala University the extensible main memory data stream management system SCSQ (SuperComputer Stream Query system, pronounces ‘sisque’) [10][11] was developed. The queries that a DSMS executes over data streams are called continuous queries (CQs). While a regular database query executes once on a finite amount of data, a CQ executes continuously until canceled.
Apache Spark is a large scale general data processing engine. It provides an API for distributed computing. It has, among other things, a component for data stream processing and a library for machine learning.

The goal with this project is to investigate how Apache Spark is suitable to be connected to a DSMS such as SCSQ and how Spark functionality can be utilized by CQs in SCSQ. The main questions for this project are:

1. How can the machine learning algorithms in SPARK be made available in CQs?
   a. In particular, it should be possible to call some of these algorithms in SCSQ queries.
2. How can the results of SPARK Streams be utilized in CQs?
   a. In particular, it should be possible to post-process as SCSQL queries the results of SPARK stream algorithms.

The questions are answered by the implementation and demonstration of a prototype system that allows Spark Streaming applications to be called from SCSQ. This implemented system is called the SCSQ Spark Streaming Interface (SSI). It is shown how SSI enables continuous clustering of received stream elements using the k-means algorithm available in the machine learning library MLlib in Spark Streaming. The cluster detection can be made either off-line by learning from a batch file or on-line where the cluster detection is continuously made in real-time.
2 Background

2.1 Apache Spark

Hadoop [16] and Apache Spark [2] are a large scale data processing engines. They provide scalability and fault tolerance based on the MapReduce [3] model. MapReduce and other similar models are very popular and useful for a wide range of applications. A drawback with Hadoop is that it relies on reading from, and writing to, stored data, which causes processing delays. This can be very inefficient for applications that rely on reusing data, for example iterative algorithms in machine learning and interactive data analysis. To efficiently support these types of applications while also providing many of the benefits of MapReduce, Spark was introduced. In [5] it is shown that Spark can run a logistic regression algorithm orders of magnitude faster than Hadoop on a big cluster.

2.1.1 Resilient Distributed Datasets, RDDs

The main abstraction used in Spark is resilient distributed datasets (RDDs) [6] stored in main-memories on a cluster. RDDs are an abstraction for distributed data and the foundation for Spark. More precisely, a RDD is a read-only partitioned dataset. A RDD can only be created from stored data or other RDDs with deterministic operations, meaning the operations must be reproducible with the same results. This is so Spark can then recompute a RDD if needed. In a Spark application you create one or several RDDs on which you can then apply different types of operations, which either transform RDDs to new RDDs or perform other actions on RDDs.
Spark supports creating RDDs from many different data sources, for example from a file on a shared filesystem such as the Hadoop Distributed File System, or from Scala/Java collections. If Spark is run locally, not on a cluster, a RDD can simply be created from a local file on the machine. Operations that create RDDs from other RDDs are called transformations. Some examples of such transformations are map, filter, and join. Other types of operations are actions, which operate on RDDs and return a value or write data to external systems. Examples of such actions are reduce, count, and countByKey. All operations on RDDs are coarse-grained, meaning the same operations are applied to every item in the dataset.

One advantage with RDDs are that they are kept in main memory. This is what makes Spark an efficient platform for iterative algorithms, since data only needs to be read from disk once, then it is kept in memory for all future iterations. RDDs also guarantee fault tolerance in an efficient way. Instead of writing to disk for fault tolerance, RDDs keep information about how they were computed. Then when a part of a RDD is lost, it can be recomputed from the last state that was saved in storage. The choice to always recompute instead of reread lost data when possible is a basic design decision with Spark.

### 2.1.2 Spark Streaming

Spark Streaming [7] is a data stream processing component built on top of Spark. Spark Streaming aims to provide for low latency data stream processing using RDDs on large clusters similar to how models like MapReduce did for off-line batch processing [8].

One problem that a large scale distributed computing system must handle is failed nodes and slow nodes [3]. In a stream processing system it is even more
important to recover from these problems quickly, since the results are delivered in real time. In Spark Streaming this is solved by having the stream processing be continuous mini-batch computations on small time intervals. The mini-batch computations are stateless and deterministic which means that the state at all times during the stream processing can be recomputed given saved input data from a previous mini-batch. This makes the recovery techniques needed to handle failed and slow nodes much simpler [8]. This model is called D-Stream and it is the main abstraction in Spark Streaming where short time-limited batch computations are performed by Spark using RDDs.

Since Spark streaming continuously runs small batch jobs rather that processing records one at the time as they arrive, it will not achieve super low latencies. This means that the approach is not be suitable for applications where fast response is needed, for example high frequency trading. This was a design decision from the beginning with Spark Streaming. The goal is to provide second or sub-second latency as that is assumed to be enough for many real world applications. When implementing a Spark Streaming application, the time interval that batch jobs are run are set as a parameter.

Since Spark Streaming is built on Spark and uses the same programming model and underlying data structure (RDD), applications can easily combine batch and stream computations. For example, stored data can be joined with streaming data.

2.1.3 MLib

MLlib [9] is a machine learning library for Spark that is released as an official module of Spark. MLib contains many well known algorithms for classification, clustering, frequent pattern mining, and more. Some examples are linear
regression, k-means, and FP-growth. Some of the algorithms are built to work with Spark Streaming such as streaming linear regression and streaming k-means. These can be both trained on and applied to streaming data. Many other algorithms can also be trained on off-line data and applied to streaming data.

In this project the k-means implementation in MLlib is used in the examples to demonstrate the implemented system SSI.

### 2.1.4 Programming model

Applications for Spark can be written in Scala, Java or Python. From here onwards Java will be used in the code examples as that is the language used in this project. A Spark application in Java is simply a Java class with a main method that uses the Spark Java API to create and operate on RDDs. The application can then be launched in Spark with the `spark-submit` script. Spark will automatically distribute the work and the RDDs to the available nodes. One thing to note is that all functions and objects needed when operating on the RDDs have to be serializable, as otherwise Spark will not be able to ship them to the nodes.

The Spark API used when implementing a Spark application is based on passing functions to the RDD operations. For Spark streaming application the case is similar, but the operations are applied to DStreams instead of RDDs. The functions are called from the main driver program, i.e. a Java class, but Spark will apply the functions in parallel on the data in the RDDs on worker nodes. In Java the functions are passed by implementing interfaces available in Spark. These can be implemented and passed to the operations in three different ways. Say we for example have a RDD called `lines` consisting of strings that we want to change to upper case by using the `map` operation. The easiest way to do this is to use the lambda syntax available in Java 8.
JavaRDD<String> linesUpperCase = lines.map(s -> s.toUpperCase());

The interface can also be implemented in an inner class. An instance of the class can then be passed to the operation.

class upperCase implements Function<String, String> {
    public Integer call(String s) { return s.toUpperCase(); }
}
JavaRDD<String> linesUpperCase = lines.map(new Sum());

The last option is to pass an anonymous inner class directly to the operation.

JavaRDD<String> linesUpperCase = lines.map(new Function<String, String>() {
    public Integer call(String s) { return s.toUpperCase(); }
});

2.1.5 Example Spark Streaming Application

Here we will see an example of a simple Spark Streaming application that reads a data stream from the local computer. The application will expect numbers in the stream and print the sum of the numbers over a 2 seconds sliding window.

First we create a SparkConf object where different settings for the application can be set. We set it to run in local mode on four cores and set the application name to WindowSum. Local mode means that Spark runs on a single Java virtual machine and not on a distributed cluster.

SparkConf conf = new SparkConf().setMaster("local[4]").setAppName("WindowSum");

13
We then create a `JavaStreamingContext` object and pass the `SparkConf` object and a time duration. This is the time interval that Spark Streaming will run batch jobs at, and thus it sets a lower limit for the latency of the application. The `JavaStreamingContext` object is the variant of the Spark-Context object seen in 2.1.4 used for Spark streaming applications implemented in Java.

```java
JavaStreamingContext jssc = new JavaStreamingContext(conf,
        Durations.seconds(1));
```

We can now create a `DStream` object containing the numbers sent over localhost:

```java
JavaReceiverInputDStream<String> lines = jssc.socketTextStream("localhost", 9999);
```

We can then use the `reduceByWindow` transformation and pass in a function that will parse an integer from each line and sum them.

```java
JavaDStream<Integer> sum = lines.reduceByWindow( new 
        Function2<Integer, Integer, Integer>() {
            public Integer call(String line1, String line2) throws Exception {
                return Integer.parseInt(line1) + Integer.parseInt(line2);
            }
        }, Durations.seconds(2), Durations.seconds(1));
```

To see the result we can use the action `print` that will print the first 10 records of every RDD in the `DStream`.

```java
sum.print();
```
2.2 SCSQ

A data stream management system (DSMS) is a system that can manage and query continuous data streams. It is similar to a database management system (DBMS), but the key difference is that it can handle data streams while a DBMS can only handle stored data. Similarly to SQL in a DBMS, a DSMS also executes high-level user-oriented queries. The queries that a DSMS executes over data streams are called continuous queries (CQs) since, in contrast to regular queries, they execute continuously until explicitly cancelled. This means that a continuous query can return an infinite stream of tuples, rather than a single table.

SCSQ (SuperComputer Stream Query processor)[10][11] is a DSMS that supports queries over high-volume distributed data streams allowing advanced computations over stream elements with low delays. In SCSQ queries are expressed in the query language SCSQL, which is an extension to the query language AmosQL [13] to work in parallel over streams. SVALI (Stream VALIdator)[12] is an extension to SCSQ provide advanced window data types that supports aggregate functions on sliding windows.

2.2.1 Amos

Amos II [1] is the system that SCSQ is built on. It is an extensible main-memory DBMS with the object oriented and functional query language AmosQL. Amos has a data model that is centered around the three concepts objects, types, and functions. All data is represented as objects in main memory. Every object is an instance of one or more types, which are used to classify the objects. Types support inheritance and can be organized as supertypes/subtypes. Functions model the relationships between objects, computations over objects, and properties of objects.
There are stored, derived, and foreign functions [14]. As the name suggests, stored functions are stored in the Amos database. They describe the properties of objects and corresponds to tables in traditional relational databases. A derived function is defined as a query over other functions. In AmosQL there is a select statement much like in regular SQL that can be used for defining derived functions. Foreign functions are important in this project as they can be used to access external systems such as Spark Streaming, so they are described in detail in 2.2.2.

The following is an AmosQL example of defining a type Car, a stored function price, and a derived function carUnderPrice that returns all cars under a given price:

```amosql
create type Car;
create function price(Car) -> Number as stored;
create function carUnderPrice(Number p) -> Car
    select c from Car c
    where price(c) < p;
```

### 2.2.2 Connecting to external systems

One way to connect to external systems from Amos and SCSQ is to define foreign functions. Foreign functions are functions that can be called from SCSQ but are implemented in some regular programming language, such as Java, C/C++, Python, or Lisp. There is for example a predefined relational database interface written in Java that can be used to connect to databases that use JDBC (Java Database Connectivity) [17].

In this project the Java interface [15] is used to connect SCSQ with Spark. The Java interface allows foreign functions to be implemented in Java and called from
SCSQ. To create a Java foreign function the signature of the function must defined as a foreign function in AmosQL, for example:

```sql
create function multiply(Number a, Number b) -> Number r
    as foreign "JAVA:MyPackage.MyClass/multiplyNumbers";
```

Here, `multiplyNumbers` is the name of the Java method that implements the foreign function. `MyClass` and `MyPackage` are the names of the class and package where the implementation of `multiplyNumbers` is located.

A Java foreign function always takes an object of class `CallContext` and an object of class `Tuple` as parameters. The tuple object is both used for input and output for the Java function. The parameters from the function called in some SCSQL query can be accessed from the tuple, and the result of the function call should be returned by updating the tuple. The `CallContext` object is used to emit the tuple back to SCSQ.

The `multiplyNumbers` foreign function is defined as follows:

```java
package MyPackage;
import callin.*;
import callout.*;
public class MyClass {
    public void multiplyNumbers(CallContext cxt, Tuple tpl)
        throws AmosException {
        double a = tpl.getDoubleElem(0);
        double b = tpl.getDoubleElem(1);
        double result = a * b;
        tpl.setElem(2, result);
        cxt.emit(tpl);
    }
}
```
As we can see in this example the two parameters to the function can be read from index 0 and index 1 in the tuple. Then the result of the function is set as index 2 of the same tuple. This is how every Java foreign function has to work. The result from a foreign function is a stream of elements returned iteratively by successive calls to the `emit()` method.
3 SCSQ Spark Streaming Interface

To connect SCSQ with Spark Streaming the data stream elements should be iteratively sent both from SCSQ to Spark and from Spark to SCSQ. To achieve this a prototype system called SCSQ Spark Streaming Interface (SSI) was implemented in this project, which allows Spark Streaming applications to be called from SCSQ. In this section an overview of SSI is given. Two examples are also shown of how SSI can be used for stream clustering with k-means implementations in Spark Streaming.

3.1 Overview

SSI makes it possible to start Spark Streaming applications from SCSQ. It also allows Spark Streaming applications to receive input stream elements from SCSQ as well as emitting result stream elements from the Spark Streaming back to SCSQ. SSI is described in more detail in section 4.1.

Figure 1 shows an overview of the different software levels in SSI. As can be seen, it allows the user to run a CQ in SCSQ and have a Spark Streaming application do the stream processing. SSI is implemented as the foreign function sparkStream() available in CQs. This function call will have the effect that a Spark Streaming application is started that will process the input data stream elements. The results of the processing, also a stream of elements, are emitted one-by-one back to the CQ by sparkStream().
3.2 K-Means in MLlib

As illustrating examples of using SSI the k-means data stream clustering algorithm implemented in Spark Streaming is called in CQs from SCSQ through `sparkStream()`. K-means is an algorithm for clustering data into $k$ number of clusters. The value of $k$ is chosen before running the algorithm. The standard k-means algorithm iteratively assigns the data points to clusters and then recalculate the cluster centers as new data points are read. Each data point is assigned to the closest cluster center and the centers are recalculated from the assigned points. Batch training is the standard way of training k-means where the training is done once on a finite set of stored data.

The k-means algorithm is available in MLlib. Furthermore, there is also a streaming k-means algorithm available in MLlib that can do on-line training where the algorithm is trained on elements that are continuously received from a stream. This means that the clusters will dynamically change as new data
elements arrive. To allow the cluster centers to continuously change with new arriving data stream elements, streaming k-means values are weighted differently depending on how long ago they arrived to the stream. The older a point is, the less weight it will have when calculating new cluster centers. How fast points decay is set by a parameter.

3.3 Off-line stream clustering

In the first example SSI is used to do off-line training of k-means with data from a training file and then on-line predictions on the trained k-means model are returned as a data stream. This means that first the k-means model is trained in Spark in an initial phase, and then a CQ receiving a stream of points can be specified that sends the points to Spark Streaming in order to determine what cluster each point should be assigned to. These points are not used to update the cluster centers as they were computed once from the training file. The clusters are thus kept unmodified while the CQ is running. The CQ will return a stream of predictions for the points.

For the off-line training of k-means, a file with 3D points, distributed over two clusters was generated. The clusters are square shaped and have their centers in (5, 5, 5) and (15, 15, 15).

For the streaming predictions a text file to simulate the elements of the real-time stream of points used by the CQ was generated containing labeled 3D points. The points’ coordinates are random, but distributed the same way as the training data. The points with even numbers as labels are closer to the center in (15, 15, 15) and the points with odd numbers as labels are closer to the center in (5, 5, 5). This was done so it could easily be seen if the predictions were correct or not. A SCSQ function called \texttt{kMeansPredictData()} was created that reads the on-line
stream with the labeled points from a file and returns them as a stream. This was done so that the data could be kept in a file, but still be used as a stream in SCSQ. The CQ will return a stream of predictions in the form of tuples of point labels associating each labeled input stream point with a cluster center to which the point was assigned.

The following CQ specifies the off-line stream clustering example as a call to the `sparkstream()` foreign function:

```sql
sparkstream("SCSQSpark.Example.StreamingKMeansPredict","2clusters.txt", 2, 20),1,kMeansPredictData());
```

The signature of `sparkstream()` is:

```scala
create function sparkStream(Charstring sparkApp, Vector sparkAppParams, Number batchDuration, Stream inputStream) -> Stream
```

It takes four parameters and returns a stream. The string parameter “`SCSQSpark.Example.StreamingKMeansPredict`” specifies the package name and class name of the application in Spark implementing the steaming function. The vector `{"2clusters.txt", 2, 20}` specifies the parameters to the Spark Streaming application, in this case the name of the file with training data, the number of clusters, and the number of iterations for the off-line training of the k-means model. The number ‘1’ is the batch duration in seconds that Spark will use as a mini-batch and `kMeansPredictData()` is a call to the previously mentioned SCSQ function that will return the elements in a file with labeled points as a stream.

The elements in the stream that `sparkstream()` returns depends on the implementation of the Spark application. In this case each element is a vector of a label, referencing the labels of the points in the input stream, and another vector, representing the cluster center the point was assigned to.
The console below shows first few results being returned from the CQ.

```
[sa.amos] 2> sparkstream("SCSQSpark.Example.StreamingKMeansPredict","2clusters.txt", 2, 20);kMeansPredictData());
[21.0,{4.65264702332497,4.85552459306029,4.50542748241488})
[0.0,{14.5345378066141,14.6072104963569,15.0.01743698944})
[1.0,{4.65264702332497,4.85552459306029,4.50542748241488})
[22.0,{14.5345378066141,14.6072104963569,15.0.01743698944})
[2.0,{14.5345378066141,14.6072104963569,15.0.01743698944})
[3.0,{4.65264702332497,4.85552459306029,4.50542748241488})
```

It can be seen that the predictions are correct, since the points with even numbered labels has been predicted to the cluster close to (15, 15, 15) and the points with the odd numbered labels has been predicted to the cluster close to (5, 5, 5).

### 3.4 On-line stream clustering

In the second example the SSI is used to do on-line training of k-means on a stream. The cluster centers will be calculated and updated from the data in a data stream from SCSQ. This means that the centers will not be static, but will change as new data arrives. They are updated with every mini-batch processed by Spark Streaming.

The input stream is a stream of 3D points produced by a CQ. Similar to the other example, a generated file with 3D points in two square clusters with centers in (5,5,5) and (15,15,15) was used, but rather than accessing the file directly to Spark it was read in a CQ by a SCSQ function called `kMeansTrainData()` that returns the points in the file as a stream whose elements are streamed to Spark Streaming to continuously produce a stream of cluster centers. For every mini-batch produced by Spark Streaming a vector with two cluster centers is returned,
since $k=2$. The cluster centers are represented as a vector containing the coordinates of the center.

The example is run with a CQ that calls $\text{sparkStream}()$, the same foreign function used in the first example. The CQ for the example is:

```java
sparkstream("SCSQSpark.Example.StreamingKMeansTrain", {3, 2, 1}, 1,
    kMeansTrainData());
```

As with the other demonstrated application the first parameter to $\text{sparkStream}()$ is the package and class name of the Spark Streaming application. The second parameter is the three parameters to the application, in this case the number of dimensions, number of clusters, and decay factor. The third parameter is the batch duration and the last parameter is the SCSQ function that reads the file with training data and returns it as a stream.

The screen shot below shows how the function is called and the first few results returned.

It can be seen that the function returns two vectors each containing three numbers. These are the current calculated cluster centers. The centers seem to converge towards the expected centers at $(5, 5, 5)$ and $(15, 15, 15)$ within a few
mini-batches. The reason that the calculated centers are very off in the beginning and then converges towards the expected centers is the on-line training of the k-means model. The model starts with random centers and then as more and more training data is streamed to the application, the centers gets updated and converges to the expected centers.

The main difference between the two demonstrated examples is that the first example uses standard batch k-means were the cluster centers are calculated once from data in a file. In the second example streaming k-means is run where the centers are dynamically updated as data arrives in a stream. This means that the first example uses both batch and streaming functionality from Spark, and the second example only uses streaming functionality.
4 Implementation

The implemented interface between Spark and SC5Q has three main components. The SC5Q receiver allows Spark to receive data streams from SC5Q, the SC5Q sink allows for the results of Spark streaming applications to be sent as a stream to SCSQ, and the Spark submitter is used to start a Spark application from SC5Q. These three components can be used all together or alone, depending on what is needed.

4.1 The sparkStream foreign function

From SC5Q, the main entry point to SSI is the `sparkStream()` foreign function. This is the function that is called to start a Spark application from SC5Q in a CQ and it will return the result as a stream sent back from the Spark application. Figure 2 shows a CQ that calls `sparkStream()` to have Spark Streaming do data stream processing. It shows an overview of how the components of SSI relate to the CQ and Spark Streaming. The small black arrows show how the components relate and the dotted red arrows show how the data streams flow. The big red arrows are the input and output stream to the CQ. When called, `sparkStream()` will create a SC5Q sink and a Spark submitter. The submitter will start the Spark application that use SC5Q receivers to get the input stream that was passed as a parameter to `sparkStream()`.

First a SC5Q Sink is started to receive data stream elements from the started Spark Streaming application. The Spark Submitter will then start Spark and run the Spark Streaming application that was passed as a parameter to `sparkStream()`. The stream provided as a parameter is sent over a socket connection to the Spark application. Since the stream from SC5Q consists of tuple objects and the corresponding Tuple class in SC5Q Java interface does not
implement Java’s `Serializable` interface, they must first be converted. The tuples are converted into `ArrayList` objects. If a tuple contains other tuples, those tuples will also recursively be converted to `ArrayList` objects. When an `ArrayList` has been sent over the socket connection to the Spark application, SSI also takes care of converting it to the desired data type in the application.

![Diagram showing a CQ calling `sparkStream()` and how the components of SSI relate to Spark Streaming and the CQ. The small black arrows show how the components relate, the dotted red arrows show how the data streams flows and the big red arrows are the input and output streams of the CQ.](image)

Figure 2: Diagram showing a CQ calling `sparkStream()` and how the components of SSI relate to Spark Streaming and the CQ. The small black arrows show how the components relate, the dotted red arrows show how the data streams flows and the big red arrows are the input and output streams of the CQ.
4.2 The SCSSparkContext class

`SCSSparkContext` (SSC) is the main entry point to SSI when implementing a Spark application to be called from SCSS. It is the class that wraps the functionality of SSI and makes it easy to use. One simply creates an SSC object and then uses that object to both receive streams from SCSS and send streams back to SCSS.

To receive a stream from SCSS the `getInputStream` method is used. It is a generic method so the data type of the stream elements must be specified, for example `Integer`, `Double` or `String`. The method receives data from SCSS to Spark as a `JavaDStream` by setting up at SCSS receiver, so it can be processed with the Spark API as desired. The `JavaDStream` will be of the same generic type as was specified when calling the method.

To return a stream back to SCSS the method `send` is used, which takes a `JavaDStream` as argument. The content of the stream will be streamed back to the `sparkStream()` foreign function. For example, to send a `JavaDStream` named `numbers` back to SCSS from a Spark Streaming application, the send method can be called as:

```
ssc.<Integer>send(numbers);
```

In this example `ssc` is the `SCSSparkContext` object defined in the application and the `JavaDStream numbers` contains elements of the type `Integer`.

As an alternative to returning `JavaDStream` objects from the Spark application, it is also possible to iteratively send stream elements back to SCSS. The same method is used but the record that is sent is passed as a parameter to `send` instead of a `JavaDStream`. This is useful if some result is calculated in every mini-batch, but is not in the form of a `JavaDStream`. Then the `send` method can simply
be called for every batch to emit results to SC SQ. For the SC SQ peer that receives the data, there is no difference if a JavaDStream was sent in a single send or if separate records are sent iteratively. To iteratively send single String element named line back to SC SQ the send method can be called several times as:

ssc.<String>send(line);

### 4.3 Supported data types

Both the methods for receiving streams from SC SQ and sending streams to SC SQ are generic, but since the data is being sent between two different systems using TCP not all types of Java objects can be sent using SSI. Only common primitive data types such as integers, doubles and strings are supported, along with some types commonly used in MLlib.

Primitive Java types are straightforwardly mapped to corresponding types in SC SQ. For example, an integer in Java corresponds to type Number in SC SQ, a String in Java corresponds to type Charstring in SC SQ, and a double in Java corresponds to type Real in SC SQ. Arrays in Java are represented by type Vector in SC SQ, so an array of Integers in Java would be a Vector of Number in SC SQ, for example \{1, 2, 3\}.

The Vector class in MLlib is a type with integer indices and doubles as values. So it is not generic as the standard Java class Vector; it only holds doubles as values. The representation of this in SCQ is simply a Vector of Real, for example \{1.1, 1.2, 1.3\}.

The LabeledPoint is an MLlib class that represents a vector with a label attached to it. The label is of type double and the vector is an MLlib Vector. This is represented in SC SQ as a Vector of (Real, Vector of Real). This structure is seen
in the first example. For example, if a *LabeledPoint* has the label 1.0 and the
*Vector* \{1.1, 1.2, 1.3\} the SCSQ representations will be \{1.0, \{1.1, 1.2, 1.3\}\}.

Lastly, the *Tuple2* class in MLlib represents a tuple with two elements. The class
is generic so the elements can have any type. However, when used in SSI the
types of the tuple must be from the supported types in SSI. The SCSQ
representation of a *Tuple2* is a *Vector of (A, B)* where \(A\) and \(B\) are different
element types. For example a *Tuple2* with a *String* element and a *Vector* element
will look like \{"Test", \{1, 2, 3\}\}.

### 4.4 SCSQ Receiver

In Spark streaming several different receivers are available so data streams can
be received from different sources. For example, receivers are available for
socket connections, file systems, and Akka actors [18]. Some more advanced
sources are also available such as Apache Kafka [19] and Amazon Kinesis [20],
but these require linking some external libraries.

Custom receivers must be implemented for Spark Streaming to receive data
streams from other sources than the ones that are already supported. To
implement a custom receiver in Spark a class that extends the abstract class
*Receiver* in Spark is implemented. The implemented receiver will handle
connecting to and receiving data from the external source, and pushing it into
Spark. The two most important methods that must always be implemented in
the receiver class is *onStart* and *onStop*, which defines what should happen when
the receiver is started or stopped, respectively.

To send data from SCSQ to Spark Streaming, a SCSQ receiver for Spark was
implemented in this project. The SCSQ receiver receives data streams that are
passed as parameters to the *sparkStream()* foreign function. The stream
elements are sent from the foreign function to Spark Streaming over socket connections as `ArrayList` objects. The receiver uses the Java class `ObjectInputStream` to receive the `ArrayList` object from the connection and for further stream processing by Spark. The `ArrayList` is thereby constructed from the SCSQ data types corresponding to the types passed to the generic method `getInputStream` that created the receiver.

### 4.5 SCSQ Sink

The SCSQ Sink is the component of SSI that receives streaming data from Spark Streaming back to SCSQ for further stream processing. The Spark Streaming application processes a data stream and then sends the result back to SCSQ as a stream. This transfer is done over sockets either locally or over a network. The SCSQ Sink can receive data from multiple parallel Spark worker nodes, which is necessary because the data computed by a running Spark application is not available on the master node, as it is distributed on the workers.

When a SCSQ Sink has been created it will continuously in a loop listen for incoming connections on the specified port. When a connection is established it will create a new thread that will run a class called `SinkWorker`. This thread will handle receiving data from Spark and emitting it to SCSQ. Thread safety was not assumed on the emit functionality to SCSQ. To make sure only one thread would emit at a time a lock object is passed from the SCSQ Sink to the threads. This object is then used in a synchronization block so that only one thread can emit at once. This means that this synchronization block is a point where data from parallel sources merge to one stream of emits to SCSQ. This type of synchronization will affect performance if the data volume is high, assuming that the data has to be sent from parallel Spark partitions to a single SCSQ. In the
current implementation there is a lock for each emitted record. This could be improved by collecting several records and then emitting the to SCSQ in a batch.

To send a JavaDStream object to external systems from Spark Streaming the generic operator foreachRDD(func) is used, where func is a function. It will apply the function func to all RDDs in the stream. It is then up to this function to push the data in the RDDs to the external system. In SSI the data is pushed to SCSQ. Internally, when send is used to send a JavaDStream to SCSQ, objects of the SCSQSinkConnection class is created, which are connection objects that connect to and send data to the SCSQ sink. The foreachPartition() operation is used to create a connection object on every partition of the RDD, but not for every record.

4.6 The Spark Submitter

The standard way to start a Spark application is to use the spark-submit script that is provided with Spark. This will launch Spark with the given settings and run the given Spark application. At the time of this project there was no supported way to start Spark programmatically instead of using the submit script. This was a problem since the intent of this project was to be able to run Spark applications from CQs executed by SCSQ. The Spark Submitter is the component of SSI that solves this problem.

With the Spark Submitter it is easy to start a Spark application from a Java program. A SparkSubmitter just has to be created and the run method called. It builds a command array and the Java Processbuilder is used to start the spark-submit script as a new process. That will start Spark and run the wanted Spark application. The SparkSubmitter will then just wait for the new process to
terminate and will have threads consuming the input and error streams of the process so it will not hang due to a pipe getting full.

4.7 Stream clustering

This section describes the implementation details of the Spark Streaming applications for the stream clustering examples seen in section 3. It is described how the k-means implementation of MLlib is used and how SSI is used to make it possible for the applications to receive and send streams to SC SQ.

4.7.1 Off-line stream clustering

The first things that must always be done when implementing a Spark application using SSI is to create a SC SQSparkContext object. The argument to the main function, *args*, and the name for the Spark application is passed as parameters. Usually when creating a Spark application, a *SparkContext* and a *SparkStreamingContext* has to be created, but they are created automatically when creating an *SCSQSparkContext*.

```java
SCSQSparkContext ssc = new SCSQSparkContext(args,
   "Predict-Streaming-KMeans");
```

The three arguments that the application expects is retrieved using the *getArg* method. Since they are passed to the application as command-line arguments to the Java class, they are of String type and must be parsed.

```java
String path = ssc.getArg(0);
int numClusters = Integer.parseInt(asc.getArg(1));
int numIterations = Integer.parseInt(ssc.getArg(2));
```
Then the training data is read from a file to a *JavaRDD*. The *getSparkContext* method is used so that the *textfile* method can be used to read a file to a RDD. The data is parsed to the *Vector* class using the map operation.

```java
JavaRDD<Vector> trainingData = sc.getSparkContext().textFile(path).map(s ->
    Vectors.parse(s));
```

Using the *train* method the k-means model is trained from the training data.

```java
final KMeansModel model = KMeans.train(trainingData.rdd(), numClusters, numIterations);
```

That is all that needs to be done for the batch off-line training of the k-means model. For the streaming part, an input stream is received as a *JavaDStream* by using the *getInputStream* method from SSI.

```java
JavaDStream<LabeledPoint> testData =
    ssc.<LabeledPoint>getInputStream(LabeledPoint.class);
```

A *JavaDStream* of predictions is then created by using the *predict* method in a map operation on the input stream. The *predict* method gives the index of the predicted cluster. That index is used to extract the correct cluster from the *clusterCenters* method. The result is that the predictions *DStream* will contain *LabeledPoint* objects with the label of the point that was used in the prediction and a vector with the cluster center that was predicted.

```java
JavaDStream<LabeledPoint> predictions = testData.map(lp ->
    LabeledPoint(lp.label(),
        model.clusterCenters()[model.predict(lp.features())])
);
```

The predictions are sent back to SCSQ using the *send* method of SSI.

```java
ssc.<LabeledPoint>send(predictions);
```
4.7.2 On-line stream clustering

Similarly to the first example, the first thing that must be done is to create a *SCSQSparkContext* object.

```java
SCSQSparkContext ssc = new SCSQSparkContext(args, "Train-Streaming-KMeans");
```

Next, the input data streamed is received to a *JavaDStream* using the *getInputStream* method in SSI. MLlib’s *Vector* is used since the k-means implementation in MLlib uses it.

```java
JavaDStream<Vector> trainData = ssc.<Vector>getInputStream(Vector.class);
```

As in the other example, the arguments to the application are parsed from the command line:

```java
int numDimensions = Integer.parseInt(ssc.getArg(0));
int numClusters = Integer.parseInt(ssc.getArg(1));
double decayFactor = Double.parseDouble(ssc.getArg(2));
```

The arguments are then used when the k-means model is created:

```java
final StreamingKMeans model = new StreamingKMeans();
model.setK(numClusters);
model.setRandomCenters(numDimensions, 0.0, 0);
model.setDecayFactor(decayFactor);
```

To train the data on the input stream the *trainOn* method is used:

```java
model.trainOn(trainData.dstream());
```

There is no method in streaming k-means that returns the cluster centers as a stream. So instead, to return the centers every batch, the *foreachRDD* operation
is used, where in every batch, the `send` method of SSI is used to emit the current cluster centers back to SCSQ.

```scala
trainData.foreachRDD(rdd -> {
    ssc.<Vector[]>send(model.latestModel().clusterCenters());
    return null;
});
```
5 Evaluation

5.1 Stream clustering measurements

When running the example CQs for off-line and on-line stream clustering, Spark monitoring functionality was used to measure the input rate of the streams to Spark as well as the processing time and delay per batch. The examples were run on a laptop with an Intel i5-450M processor and 4GB of RAM.

5.1.1 On-line stream clustering

Figure 3 shows the input rate of the stream with training data. As can be seen, the average rate is around 1,000 events per second. An event in this case is simply a received data point.

Figure 3: Input rate for the training data to the streaming k-means with on-line training application.

Figure 4 shows the processing time per batch of the application. Since the application was configured to run in 1 second batches the processing time would ideally always be under 1 second.
In addition to processing time, other things also impact how fast a batch will be handled in Spark Streaming. One example is the scheduling delay which is how much time it takes for Spark to submit the jobs of a batch. In figure 5 the total delay per batch is shown, this is the total time it took Spark Streaming to handle each batch. As can be seen, this also closely follows the curve of the processing time, meaning that no other factor impacted the total delay significantly.

5.1.2 Off-line stream clustering

Figure 6 shows the input rate for the stream of points used for prediction.
As can be seen the input rate varies with an average of about 400 events per second.

![Input rate graph](image)

**Figure 6**: Input rate for the points for prediction to the streaming k-means with off-line training application.

Figure 7 and 8 shows the processing time and total delay, respectively. Also in this application the curves for processing time and total delay follows each other, meaning no other factor than the processing time had a significant impact on the delay.

![Processing time graph](image)

**Figure 7**: Processing time for the off-line stream clustering example.
5.2 Interface performance

When using SSI to run Spark Streaming applications from SCSQ, the main decider of the performance is the performance of the processing done in Spark. This depends mainly on the hardware Spark runs on, Spark itself, and the implementation of the application. This is independent of the SSI implemented in this project, but some basic testing was made to investigate the performance of SSI and to locate any possible bottlenecks. Since SSI handles input and output from Spark applications when run from SCSQ, the tests was on the performance of communication of data using SSI.

5.2.1 SCSQ Receiver performance

To test the performance of the SCSQ receiver a Spark Streaming application that counts the number of records received from SCSQ was implemented. On the SCSQ peer that sent the stream to Spark the heartbeat function was used to generate a stream of numbers. A separate small Java program that uses the SCSQ Java interface to execute the same function on a SCSQ peer and then count the number of records read per second was also implemented. Both the Spark
application and the Java program was run for 250 seconds. In table 1 the average number of records per second over the entire run of 250 seconds is shown for the SCSQ receiver and the Java program.

<table>
<thead>
<tr>
<th>SCSQ Receiver:</th>
<th>7102.4 records/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java Program:</td>
<td>7312.8 records/s</td>
</tr>
</tbody>
</table>

Table 1: Average number of records per second for the SCSQ Receiver and the Java program.

As can be seen, the rate for the SCSQ receiver and the Java program does not differ significantly. This means that the bottleneck when sending data from SCSQ to Spark is not the receiver in SSI, but instead the Java interface to SCSQ.

5.2.2 SCSQ Sink performance

Since the SCSQ sink gathers data from many different threads in parallel, one from each partition in Spark, and then emits it as a single stream to SCSQ one-by-one it was investigated if that caused a performance bottleneck for the SCSQ sink. To test this a Java program that opens a connection to an SCSQ sink, using the `SCSQSinkConnection` class, and sends data as fast as possible was implemented. A SCSQ sink was opened via a foreign function in SCSQ. Multiple instances of the Java program were then started so that they would send data to the SCSQ sink in parallel. The average number of records handled by the SCSQ sink per second was then measured for one, four and ten parallel instances of the Java program. Table 2 shows the results.
Table 2: Average number of records per second received by a SCSQ Sink from 1, 4 and 10 parallel sources.

<table>
<thead>
<tr>
<th>Number of Instances</th>
<th>Average records per second</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>79640.8 records/s</td>
</tr>
<tr>
<td>4</td>
<td>77199.0 records/s</td>
</tr>
<tr>
<td>10</td>
<td>78679.9 records/s</td>
</tr>
</tbody>
</table>

As can be seen the average number of records per second is very similar for one, four and ten parallel instances of the Java program. This means that reading from several parallel connections does not create a bottleneck for the SCSQ sink, at least up to 10 parallel instances. Since the records are emitted one-by-one with a synchronization lock, the overall performance is probably still effected by that. This could be improved by buffering records at the sink.
6 Conclusions

The main questions posed for this project were all related to how Spark Streaming could process data stream algorithms called in CQs processed by SCSQ. It was also specifically asked how machine learning algorithms in Spark could be made available in CQs. From the start of the project is was concluded that these questions all required that it must be possible to connect SCSQ with Spark Streaming and send data streams both ways. It must also be possible to start Spark Streaming applications from SCSQ.

SSI can start Spark Streaming applications from a simple foreign function call in a CQ executed by SCSQ. The examples showed how machine learning algorithms in Spark Streaming can be accessed in the CQs. SSI makes it easy to implement the foreign function that iteratively sends stream elements from SCSQ to a dynamically started parallel Spark Streaming algorithm which processes them as DStream objects. The result of the parallel processing is returned back from Spark Streaming to SCSQ either as a Dstream object or iteratively element-by-element. As a conclusion to this we can say that SSI succeeds in answering the two questions for this project.

In the basic investigation that was made into performance it was concluded that the bottleneck for sending data back to SCSQ from Spark Streaming is the emission of elements one-by-one. For between one and 10 parallel sources sending data to the SCSQ sink, no difference in performance was seen. Buffering elements in the SCSQ sink could possibly alleviate this.
7 Future Work

More of the Spark Streaming algorithms available in MLlib can be made available to SCSQ by SSI. Right now SSI is built for scenarios with streaming data and Spark Streaming. It would probably not be that hard to modify SSI so that it works just as good with off-line batch applications in Spark. This would make it easy to also make Spark batch functionality available to SCSQ. One more thing that could be changed in SSI is the SparkSubmitter. When starting this project there was no way to programmatically start Spark, and thus the Spark submitter had to be implemented. However, towards the end of this project the developers behind Spark released a Spark Launcher library that solves the same problem. Since that library is maintained with Spark now, the Spark Submitter should probably be changed to use that library.

It could be interesting to in the future compare the performance of one or several algorithms implemented in both Spark and SCSQ. This would then have to be tested on a variety of different hardware to see if different hardware configurations benefit the two platforms in different ways. Researching this could answer the question on when it is beneficial to implement something in Spark and use SSI to make it available to SCSQ instead of implementing it directly in SCSQ.

SCSQ has the ability to run CQs over distributed streams. It can split a stream into many parallel sub-streams and have the queries execute in parallel over them [21]. This functionality could be used in SSI to improve the performance. When a CQ is run in SSI every parallel instance of SCSQ could then connect to a partition in Spark Streaming. This would avoid the problem of having result stream elements from many Spark partitions being sent to a single SCSQ, and the discussed problem of having a synchronization lock on the emit.
Appendix A

Java and SCSQL code for off-line stream clustering

Spark Streaming application implementing streaming k-means with off-line training:

```java
package SCSSpark.Example;
import org.apache.spark.api.java.*;
import org.apache.spark.streaming.api.java.*;
import SCSSpark.*;
import org.apache.spark.mllib.linalg.*;
import org.apache.spark.mllib.regression.LabeledPoint;
import org.apache.spark.mllib.clustering.*;

public class StreamingKMeansPredict {
    public static void main(String[] args) {
        SCSSparkContext sc = new SCSSparkContext(args,
                "Predict-Streaming-KMeans");
        String path = sc.getArg(0);
        int numClusters = Integer.parseInt(sc.getArg(1));
        int numIterations = Integer.parseInt(sc.getArg(2));

        JavaRDD<Vector> trainingData = asc.getSparkContext().textFile(path).map(s ->
                Vectors.parse(s));
        final KMeansModel model = KMeans.train(trainingData.rdd(), numClusters, numIterations);

        JavaDStream<LabeledPoint> testData = sc.<LabeledPoint>getInputStream(LabeledPoint.class);
        JavaDStream<LabeledPoint> predictions = testData.map(lp ->
                new LabeledPoint(lp.label(), model.clusterCenters()[model.predict(lp.features())]));

        sc.<LabeledPoint>send(predictions);
        sc.start();
        sc.awaitTermination();
    }
}
```
SCSQ function that reads the prediction points file and returns it as a stream:

```scala
create function kMeansPredictData() -> Stream as
   streamof(select [v[0], [v[1], v[2], v[3]]) from vector v where v in csv_file_tuples("kMeansData/labeledpoints3d.txt");
```

The CQ for the example:

```sql
sparkstream("SCSQSpark.Example.StreamingKMeansPredict","trainData.txt", 2, 20),1,kMeansPredictData());
```

An extract of the `predictData.txt` file:

```
0.0,19.212674181065,16.50904928083291,13.601707726677265
1.0,9.410593154369712,6.87945770862373,0.37999594788539315
2.0,10.977809122664953,11.8947242838393298,17.65821187343263
3.0,2.848642930158344,8.603880549810707,2.584493o528486112
4.0,19.0285822522858223,17.308054467395976,17.419567844022446
5.0,8.314895882692307,9.256802773474739,2.785888401713916
6.0,10.4763003300258,18.488956119592597,16.20979221277759405
7.0,7.8132675309087,4.995811316224241,5.910163695349851
8.0,17.6810308218979,14.89632217427369,11.984592354659632
9.0,6.76355470251766,7.9575363351074,8.620728342465181
10.0,10.01591219203662,16.437784062059666,17.94738944387541
11.0,8.70367552443555,1.2851773849721637,9.80688808731573
12.0,19.0110344078629,10.0093227696065,14.851047679132812
13.0,9.804532465342247,4.4801153194345338,5.64565563195706
14.0,14.0986010879876,11.437289087844705,14.156657673534538
15.0,5.973574450660353,2.854560202685459,6.82252676610582
16.0,15.3753343731815487,15.911169051669427,17.461474517919452
17.0,8.412913866379107,2.782563692952866,9.607904978012
18.0,19.90890193824294,11.819175656335778,17.919993007906776
19.0,5.431478334607103,0.8206445394491269,5.489690969009913
20.0,12.237570005391747,14.294704624772276,19.5132296433246
```
Appendix B

Java and SCSQL code for on-line stream clustering

Spark Streaming application implementing streaming k-means with on-line training:

```java
package SCSQSpark.Example;
import org.apache.spark.streaming.api.java.*;
import SCSQSpark.*;
import org.apache.spark.mllib.linalg.Vector;
import org.apache.spark.mllib.clustering.*;
public class StreamingKMeansTrain {  public static void main(String[] args) {
    SCSQSparkContext ssc = new SCSQSparkContext(args, "Train-Streaming-KMeans");
    JavaDStream<Vector> trainData = ssc.<Vector>getInputStream(Vector.class);
    int numDimensions = Integer.parseInt(ssc.getArg(0));
    int numClusters = Integer.parseInt(ssc.getArg(1));
    double decayFactor = Double.parseDouble(ssc.getArg(2));
    final StreamingKMeans model = new StreamingKMeans();
    model.setK(numClusters);
    model.setRandomCenters(numDimensions, 0.0, 0);
    model.setDecayFactor(decayFactor);
    model.trainOn(trainData.dstream());
    trainData.foreachRDD(rdd -> {
        ssc.<Vector[]>send(model.latestModel().clusterCenters());
        return null;
    });
    ssc.start();
    ssc.awaitTermination();
    }
```
SCSQ function that reads the training data and returns it as a stream:

```sql
create function kMeansTrainData() -> Stream as streamof(csv_file_tuples("trainData.txt"));
```

The CQ for the function:

```sql
sparkstream("SCSQSpark.Example.StreamingKMeansTrain",[3, 2, 1],kMeansTrainData());
```

An extract of the `trainData.txt` file:

```
13.990653152798863, 15.12247263610114, 13.894068946384797
6.613648164178274, 0.1300705525675505, 0.1272279485132255
16.78860382108096, 17.20413884821058, 10.571017547143192
4.011348259454936, 5.943704998152314, 8.670099400848866
16.353578436602138, 14.30038867417635, 17.64623577066373 8.183927403677906
0.766451586294785, 9.240310907627146
17.39470645636475, 16.080151574776913, 16.892918200037215 4.770959746154029
5.962624872683264, 1.5569691073764613
14.08316079690525, 19.951831799914252, 11.072817563584529
1.6166998219388873, 6.138647046438806, 5.3962704759399253
16.34074082840101, 11.868508480809147, 12.442026059577666
1.2328497898944283, 0.24832211791708803, 6.6588812972383
19.525380480643165, 13.38829567096314, 11.85287624357857
8.393037192509063, 4.642125543748428, 9.573614515252416
16.48170305057186, 10.648406916684829, 14.792561212303749
9.257358707245572, 4.691713380412816, 3.2949232054879965
14.350356049815243, 15.119126699702646, 19.553874144836826 1.708785629328264
1.9470061502081566, 7.0064314799372
12.5004375809991438, 15.359005266884779, 12.59356456199221
3.2522501902779366, 6.786730469593328, 2.536739521806694
```
References


